Experiences with Extension Programming and Scripting in Python

Charles Anderson
IEOR Department, UC Berkeley
cander@ieor.berkeley.edu

Abstract

Our experiences interfacing Python with optimization software are presented. The same work was done using extension programming and scripting. Due to the nature of the C libraries we were using, we found problems related to memory corruption and C namespace collisions when we used the libraries as Python extensions. For our application and the software we are interfacing to, we found that using the libraries as standalone programs connected via Python scripting was a better way to use the optimization codes via Python.

1 Introduction

We are developing a toolbox or workbench for researchers in the field of Operations Research. We want to build a simple environment that can be used by researchers that does not require programming in C. It should be possible to add new tools to the toolbox with little or no C programming. We need to support new file formats easily. Ideally, we would also like to make this available with multiple user interfaces: script files, command line, WWW, window systems such as X-Windows.

Obviously, Python is a good choice. The syntax is easy to learn. There are a large number of existing libraries. It can be easily extended to add new functionality written in C or Python. The interpreted nature makes for very rapid development, and programmers are freed from low-level, error-prone details such as memory management.

Given that these solver codes must be written in C (or Fortran) for performance, our first impulse was to use extension programming where we added a Python interface to existing C code via some C-language glue. We tried this, and although it worked, we encountered some problems that made this a less than ideal solution.

Our second implementation was based on simpler ideas of creating Python wrappers around existing applications and files. At the core, these are like more traditional scripts such as shell scripts; they execute programs to read and write files. However, we encapsulate these details to create a friendlier, object-oriented environment.

Our contribution is an experiment in which we integrated tools using both extension and script programming. Due to the nature of the software we were working with, we found scripting to be more effective than extension programming.

2 Background

Our main work is in the field of combinatorial optimization. Our particular area of research is known as *Network Flows* [AMO93]. The basic problems to be solved are fairly easy to explain, especially to people with even the most basic knowledge of graph theory. For example, a maximum flow problem involves a network or graph of nodes connected with arcs[1]. Each arc has a maximum capacity. The objective is to push as much flow from a source node through the network to a sink node without exceeding the arc capacities and without leaving any surpluses or deficits at any nodes. A physical example would be moving fluid or gas from one location to another through a network of pipelines. Sometimes we may work with subproblems of network flows, and other times we may use more general techniques such as linear programming.[2]

For network flow research, we have the following entities:

- Problem Instances are instances of problems we wish to solve

[1] The following terms are often synonymous and used interchangeably: network/graph, arc/edge, and node/vertex.

[2] For the remainder of the paper, we will assume that we are only working with network flow problems.

- Problem Generators are used to create random instances to test with
- Solvers are the optimization software used to solve instances
- Translators transform an instance from one format to another
- Solutions are the result of a solver working on an instance
- Validators are code to compare solutions or to validate that solutions are correct

The figure below shows the relationships between these entities:

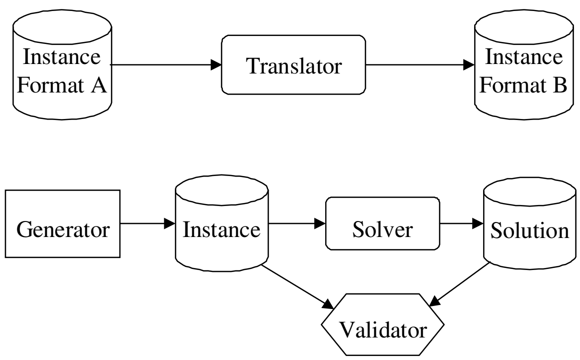

Figure 1: Entities in the Toolbox

Often a user has one or two instances of a problem to solve. Sometimes this is a trivial problem where the user wants to try out a simple example or a student's homework problem. In this case, having a simple, interactive tool is very desirable. For more serious work, we wish to solve fairly large instances with thousands or tens of thousands of nodes where the number of arcs is at least an order of magnitude larger.

Other times we wish to try multiple algorithms to compare solutions or performance on one or more benchmark problem instances that tend to be large to create meaningful timing results. The algorithms would be run on the same instance, but the instance may be in different formats for the different solvers. Regardless, a problem instance is typically read from a file into memory, the solver computes a result and the solution is written to another file

2.1 Problems Facing Researchers

There are many commercial and public domain software packages to solve such problems. New algorithms are published and often implemented by researchers. The first problem is that often the implementations are not particularly high quality, especially compared to commercial software packages. Another problem is that although there are some standard formats for problem instances, many tools have their own input and output formats, as well as strange and sometimes non-obvious usage. The problems can be compounded by 'real world' applications where the initial data is not in a standard format used by any of the tools but rather a format specific to the application. Basically, we lack a uniform, high quality set of software tools.

This Tower of Babel problem is exacerbated by steep learning curves for some tools. Often users are required to write C code to use the tools, and few researchers are particularly proficient in C. All of these obstacles discourage users, force them into using only one tool because it is all they know, or cause them to walk away from a given problem out of a feeling of frustration especially if they just wanted a quick answer to a simple problem.

2.2 Related Work

We first became aware of the concept of extension programming in the context of Tcl [Oust94]. There are many examples of extension programming projects. Dubois has been providing extensions to existing physics codes with a custom language [Dub94] as well as with Python [YDM96]. Commercial products such as Matlab also provide a high-level script language to high performance implementations of matrix manipulation code. Matlab can be extended by writing scripts or linking in C or Fortran code.

Shell scripts [Prat85] are an old idea to coordinate the execution of programs and link their input and output streams together. The beauty of command pipelines is often held up as an example of software reuse technology that actually works [Cox86]. Our work creates higher level abstractions on top of programs and files.

The Expect extension [Lib90] to Tcl provides a useful facility to control interactive programs from extended Tcl scripts. The tools we use are not typically interactive, but in some cases we may need to use such a tool, in which case we can use the Python version of Expect to interface with the tool.

TkMan [Phel95] is an example of using an external C program to perform compute-intensive tasks instead of using extension programming. This is very much like what we do, but we wish to create a framework in which we can plug in many such programs rather than just one or two that perform specific tasks.

Another way to view this work is in terms of wrapping 'legacy applications' for use in new systems. Our applications are not necessarily old, but they do share some common traits: existing code that cannot be rewritten to conform to a new environment, the desire to maintain an arms-length relationship with the code, and possibly even code that must run on dedicated hardware.

3 Extension Programming

The initial project planned to put a Python front-end onto libraries of existing code. This would allow researchers to create or convert problem instances using the higher level tools available to the Python programmer and then invoke a solver written in C. The solution could then be analyzed or saved from the Python environment. A casual user would never have to write C code, and someone wishing to extend the toolkit would have to write a minimum of glue-code in C, but this code would only have to be written once.

We wanted to move the I/O functions from C code to Python where it can be much easier to read and write various strange file formats. Given *M* solvers and *N* file formats, we wanted to change the problem of supporting all file formats for all solvers from *MN* software tasks to *M+N* tasks.

It seemed the key was to develop a common representation of problems (i.e. networks) to be shared between the Python code and the solvers. We considered the idea of some kind of least common denominator representation for graphs that would be shared by all solvers. This was quickly discarded because no one data structure can be all things to all solvers, this would only work in cases where we had the source code for the solver, and it would involve significant modification to the algorithms.

Our second idea was to use the common representation at a layer above the solver codes. The Python code would manipulate the common representation, the glue logic would translate from the common representation to the 'native' representation used by the solver, and the solver would continue to use its original/native representation. Similarly, results (output) would be translated from the native representation to the common representation where they would be available to the Python code.

The figure below shows the second design where the glue code translates between the common and native formats.

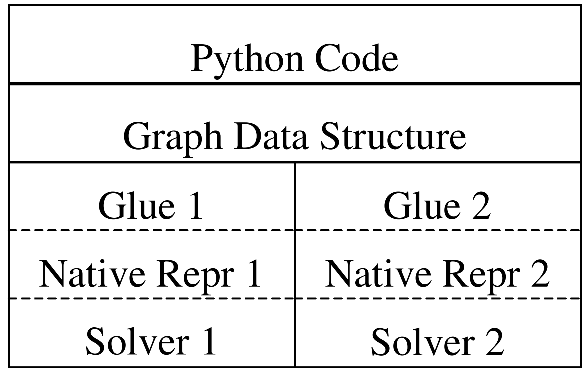

Figure 2: Relationships between the representations and the solvers.

This common data structure was designed to be simple. The objective was to make it easy to translate between the common structure and the various native formats. It was never a requirement that this representation be particularly efficient for implementing algorithms. Therefore, we did not consider using more sophisticated data structures such as the *kjBuckets* package from Aaron Watters [Watt], but rather we created a simple representation with an array of nodes and an array of arcs. The simple interface is shown below:

CgGraph()	Create a new, empty graph
getSource setSource	Get or set the source node in the graph
getSink setSink	Get or set the sink node in the graph
addArc getArc numArcs	Add an arc, get an arc, or get the number of arcs in the graph
getNode setNode numNodes	Add a node, get a node, or get the number of nodes in the graph

Notice that the interface lacks methods to get all the arcs into or out of a node or to get all of the neighbors of a node. Those are the types of operations that a graph data structure used for a solver would need, but since this structure is not used directly by the solvers, we do not have such operations.

We implemented the common graph representation and adapted an existing maximum flow solver (called GOLD [Bad91]) to work as a Python extension module. Basically, the work involved replacing the file I/O routines in the standalone version of GOLD with fairly simple glue code to translate the common graph representation.

Some of this work to interface the C code with Python can be automated using a tool such as SWIG [Beaz96]. However, the bulk of the work involved developing the code to translate between the common graph representation and that used by GOLD. This type of translation code cannot be automated using a tool like SWIG.

With our extended Python interpreter, we were able to define graphs from Python by adding nodes and arcs, and compute the maximum flow for a network. Although this worked for simple test cases, we found some problems during testing and made some observations based on the implementation.

- When the extension code was ported from FreeBSD to SunOS, the extended Python interpreter began crashing mysteriously, especially when instances of the common graph were deleted after the solver was run. After some effort, we found that the GOLD solver contained a number of small bugs and one major off-by-one array bounds error. This was corrupting the heap in such a way that when running the interpreter under SunOS it crashed, but it did not crash under FreeBSD.
- This scheme requires two representations of the graph be present in memory at the same time – i.e. it doubles memory usage. During a translation phase, both representations may be accessed simultaneously meaning that we cannot assume one representation would be paged out. This would lead to virtual memory thrashing for large problem instances much sooner than if we only had a single copy in memory.
- In the longer term, if we were to add more solvers, it would become increasingly likely to have C-language namespace collisions when linking together multiple solvers. Although we can work around this in cases where we have source code, it would be very difficult to fix in cases where we only have a binary library.
- Although the C programming was fairly easy once we became familiar with extension programming for Python, it still appeared that it would be rather daunting for novice programmers who wanted to add a new solver to the toolkit, especially since the translation code cannot be automated with tools like SWIG.

Basically, although this scheme worked for a single solver (once the memory bugs were fixed), it seemed that we would continue to have problems as we added new solvers.

4 Wrapping and Scripting

Despite the many problems with our solver codes, there were some useful characteristics. The solvers were already available as standalone programs. Also, the user's interaction with the solvers tends to be limited the to the beginning and end of the process between which the solver runs without user intervention.

Therefore, we began to look at using Python more as a shell-scripting language. We would leave the solvers as standalone programs and allow them to read and write their own native file formats on disk. Then we could use Python to develop file conversion filters and as a tool for coordinating the activities.

We created a framework of base classes in Python to represent solvers, instances, solutions, etc. These classes interact to solve user problems. In addition to base classes, there are sub-classes for particular solver or file types. There are common operations and attributes for base classes as well as specific operations and attributes for the sub-classes. For example, solvers all have a `solve` method that works on an instance and creates a solution. A sub-class might have an attribute to control the verbosity of the output generated in the solution, whereas others might not. Similarly, the output of a solver, a solution instance, may contain data specific to the algorithm that generated it in addition to the information that would be common to all solutions. A subset of the class hierarchy is shown in the figure on the next page.

The Python classes we developed for the sub-classes are user-friendly ‘handles’ to the real instances that are typically represented in the file system. They provide a uniform look-and-feel for the disparate tools and objects. We have a common graph representation written in Python to facilitate translations. To convert from one file format to another, the user can write Python code to read the input format and build the graph in memory. Then the internal representation can be written out via more Python code into a file in another format. Therefore, we only need to write $M+N$ translators rather than MN translators.

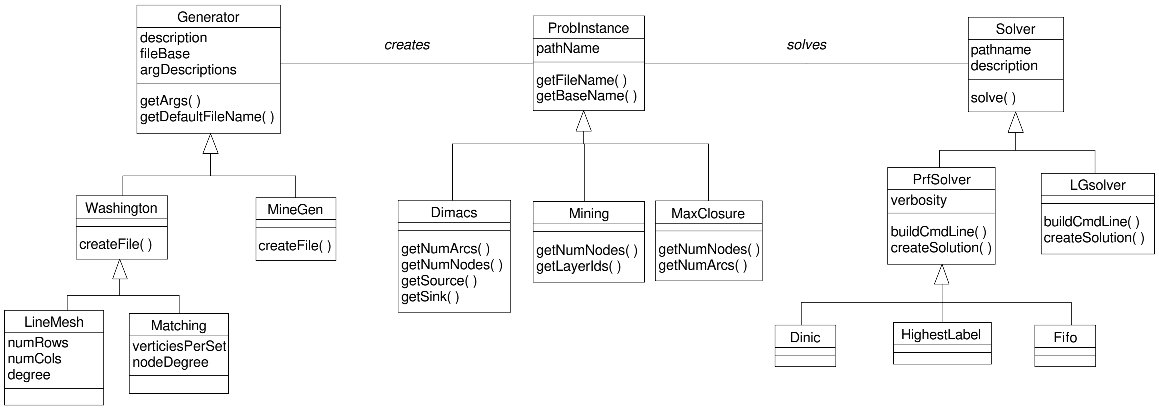

Figure 3: Generators, Instances, and Solvers

At the moment, the classes are completely implemented in Python without any use of extensions. To add a new solver to the toolbox, a user writes Python code to sub-class an existing solver class and adds the particular methods and attributes for the new solver. If we found a solver that was reliable (and free of memory bugs), and if there were some strong reason to include it into an extended Python interpreter, there is nothing in this architecture that precludes that. Like any extension, it would be pretty seamless when viewed from the Python code.

Obviously, external solvers imply the need to create a new process to run the solver. Since most of our problems are fairly large and require tens of seconds, tens of minutes, or more to solve, the overhead of a new process is minimal.

Below are some examples of using generators, solvers, solutions, and instances.

```
# create an instance
import GeneratorPrograms
mesh = GeneratorPrograms.Mesh()
mesh.numRows = 10
mesh.numCols = 20
mesh.capacity = 100
file = mesh.getDefaultFileName()
inst = mesh.createFile(file)
# solve it with two solvers
import Solvers
dinic = Solvers.Dinic()
fifo = Solvers.Fifo()
soln1 = dinic.solve(inst)
soln2 = fifo.solve(inst)
# print info about the solutions
print 'Flow values:', \
  soln1.flowValue, soln2.flowValue
print 'Solving times:', \
  soln1.solveTime, soln2.solveTime
# retrieve an instance from disk
import ProbInstance
inst = ProbInstance.getInstance \
       ('mesh-10-20-100.max')
print 'Instance dimensions:', \
  inst.numRows, inst.numCols
```

5 Status and Future Work

The toolbox currently includes the following components:

- Five solvers - three for general maximum flow and two for a mining application.
- Over a dozen generators - mostly for maximum flow, but also a few mining problem generators.
- Three Problem Instance types and three Solution types.
- Two Instance Translators.
- One Validator.

At the current time the software is still evolving rapidly, so the user community is limited to only a couple researchers. Using a very early version of the framework, we were able to find more errors in the GOLD solver. We generated some problem instances, ran GOLD, and used the Validator to verify the solution. We found that it was violating some of the problem constraints. Hence, we have stopped using GOLD and no longer support it in the toolbox.

Our current prototype uses a text terminal based interface built from components in [Lutz96]. We plan to develop a GUI, probably using *TkInter* to allow browsing objects and tools. Using the toolbox from the Web will require writing translators read and write HTML – i.e. new sub-classes of Instance and Solution. We plan a more general release once we have one of these interfaces available.

Since the Python wrappers provide an arms-length connection to the objects and tools, we plan to investigate accessing tools and objects remotely over a network. We are considering ILU [JS96] to control remote tools and instances via a Python/ILU server running on the remote machine and an ILU client process on the local machine. This will allow a researcher to use the workbench on a personal workstation to control solvers running on 'big iron' machines or an entire network of workstations (NOW) supercomputer cluster.

Since some researchers start with problem instances in Excel or wish to post-process solutions with Excel, we would like to use the OLE/COM [Brock95] support in *PythonWin* to build a bridge between Excel and the solver workbench.

6 Conclusions

We have used Python to create a toolbox to bring together many existing optimization solver programs. We first tried extension programming but met with limited success. We encountered a number of problems mostly related to the type of software we were attempting to interface with.

Fundamentally, there is nothing wrong with extension programming, but it appeared that it would not scale for our application. The problems are mostly related to the eclectic collection of software we are trying to use:

- There is no common architecture to the software tools – they are not a simple collection of APIs in a well-designed library.
- The software comes from many sources, which can lead to integration problems.
- We are attempting to interface to non-professional software that may be buggy, and it is more apt to have integration problems.
- Although the creation of C-language interface logic can be automated via tools like SWIG, the bulk of the interface involved translating graph representations which could not be automated.

Therefore, we fell back to more traditional shell-script-like programming wherein the solvers are standalone programs that are invoked to read and write problem and solution data on disk. We created Python object wrappers around these tools to provide an easy to use interface.

Acknowledgements

Thanks to Eli Olinick whose travails were the initial seeds of this project. Thanks to Mark Lutz for his Python examples that hammered home the concepts of a framework. The author is indebted to the anonymous reviewers for their comments and suggestions. And of course, thanks to Guido for Python, without which this would probably have been a Tcl project.

References

AMO Ahuja, Ravidra, Magnati, Thomas, and Orlin, James. *Network Flows: Theory, Algorithms, and Applications*. Prentice-Hall, 1993.

Bad91 Badics, Tamas. An implementation of Goldberg's push-relabel algorithm. Source code available at: `ftp://dimacs.rutgers.edu/pub/netflow/maxflow/solver-5`

Beaz96 Beazley, David, Using SWIG to Control, Prototype, and Debug C Programs with Python. In *Proceedings of the Fourth International Python Conference*. 1996.

Brock95 Brockschmidt, Kraig. *Inside OLE: Second Edition*. Microsoft Press, 1995.

Cox86 Cox, Brad. *Objected Oriented Programming: an Evolutionary Approach*. Addison-Wesley, 1986.

Dub94 Dubois, Paul. Making Applications Programmable. *Computers in Physics*, 11 (1) 1994, pp. 70.

JS96 Janssen, Bill and Spreitzer, Mike.. *ILU 2.0aplha8 Reference Manual*. XEROX Palo Alto Research Center, March 1996

Lib90 Libes, Don, "expect: Curing Those Uncontrollable Fits of Interaction". In *Proceedings of the Summer Usenix Conference*, pp. 183-192.

Lutz Lutz, Mark. *Programming Python*. O'Reilly and Associates, 1996.

Oust94 Ousterhout, John. *Tcl and the Tk Toolkit*.

Addison-Wesley, 1994.

Phel95 Phelps, Thomas. "Two Years with TkMan: Lessons and Innovation, Or Everything I Needed to Know about Tcl/Tk I Learned from TkMan", In *Proceedings of the 1995 Tcl/Tk Workshop*, Usenix Association.

Prat85 Prata, Stephen. *Advanced Unix – A Programmer's Guide*. Howard W. Sams & Co., Inc, 1985.

Watt Aaron Watters, *kjBuckets* extension, Python contributed software, `http://www.python.org/ftp/python/contrib/Math/kjbuckets.tar.gz`.

YDM96 Yang, Tser-Yuan, Dubois, Paul, and Motteler, Zane. Building a Programmable Interface for Physics Codes Using Numeric Python. In *Proceedings of the Fourth International Python Conference*. 1996